John von Neumann Institute for Computing

Generic Locking and Deadlock-Prevention with C++

Michael Suess, Claudia Leopold

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 211-218, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Generic Locking and Deadlock-Prevention with C++

Michael Suess and Claudia Leopold

University of Kassel, Research Group Programming Languages / Methodologies
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
E-mail: {msuess, leopold}@uni-kassel.de

Concurrent programming with threads heavily relies on locks. The locks provided by most threading systems are rather basic and suffer from a variety of problems. This paper addresses some of them, namely deadlocks, lack of exception-safety, and their procedural style. We describe higher-level locks that can be assembled from the more basic ones. Throughout the paper, we refer to C++ and OpenMP for parallelization, but most of the functionality is generic and independent from OpenMP. This work is part of the AthenaMP project.

1 Introduction

Locks are one of the most important building blocks of concurrent programming today. As with the advent of multi-core processors, parallel programming starts to move into the mainstream, the problems associated with locks become more visible, especially with higher-level languages like C++. This paper addresses the following ones:

- lock initialization and destruction is not exception-safe and very C-ish. Lock destruction is forgotten frequently.
- setting and releasing locks is not exception-safe and C-ish, as well. Unsetting Locks may be forgotten in complicated code-paths with a lot of branches.
- deadlocks are possible when using multiple locks

To solve the first two problems, a common C++ idiom called *RAII* is used. RAII stands for *Resource Acquisition is Initialization*[1] and combines acquisition/initialization and release/destruction of resources with construction and destruction of variables. Our solution to the first problem is called lock adapter and has the effect that locks are initialized and destroyed in constructors and destructors, respectively. Our solution for the second problem is already well known as guard objects or *scoped locking* and means that locks are set and unset in constructors and destructors, respectively.

The third problem is solved in two ways: first by extending the guard objects to multiple locks and internally choosing the locking order in a deterministic way, and second by introducing so-called leveled locks that enable the creation and automatic control of a lock hierarchy that detects possible deadlocks at runtime.

The generic locking functionality presented here is part of the AthenaMP open source project[2]. Its main goal is to provide implementations for a set of concurrent patterns (both low-level patterns like advanced locks, and higher-level ones) using OpenMP and C++. These patterns demonstrate solutions to parallel programming problems, as a reference for programmers, and additionally can be used directly as generic components. The code is also useful for compiler vendors testing their OpenMP implementation against more

involved C++-code, an area where many compilers today still have difficulties. A more extensive project description is provided by one of the authors in his weblog[a].

The term *generic* warrants some more explanations at this point. From all the functionality introduced in this paper, only the adapters are tied to a particular lock type as provided by the parallel programming system (PPS). New adapters are trivial to implement for different threading systems, and we have done so as a proof-of-concept for POSIX Threads. As soon as these adapters exist, higher-level functionality built on top of the adapters can be used. A slight deviation from this rule are levelled locks — they have a configurable backend to actually store the level-information, and one of these backends uses thread-local storage as is available in OpenMP. To make up for this, an otherwise identical backend has been implemented that is as generic as the rest of the components.

Sect. 2 describes the generic lock adapters and guard objects. With this infrastructure in place, Sect. 3 concentrates on the important problem of deadlocks. Benchmarks to evaluate the performance penalties associated with the additional functionality provided are given in Sect. 4. Some implementation problems we had are shown in Sect. 5. Related work and our contributions are highlighted in Sect. 6, a summary closes the paper in Sect. 7.

2 Generic Locking

In this section, we introduce the basic locking constructs provided by the AthenaMP library in detail, along with short examples on how to use them. The first class of locking constructs (called lock adapters) is described in Sect. 2.1. The concept of *scoped locking* is applied to these adapters in Sect. 2.2, leading to guard objects.

2.1 Lock Adapters

A lock adapter is a simple wrapper-object for a traditional lock as found in most threading systems. In AthenaMP, lock adapters are provided for the OpenMP types `omp_lock_t` and `omp_nest_lock_t`. The interface for an `omp_lock_ad` is shown in Fig. 1, along with a simple example of how to use them in Fig. 2. The interface should be self-explanatory, possibly except for the `get_lock`-method. It is useful, if you need to call library routines that expect the native lock type, as explained by Meyers[3]. In the example, a simple bank transfer function is sketched (stripped to the bare essentials). The locks are encapsulated in an `account`-class in this case, each account has its own lock to provide for maximum concurrency.

Adapters for other lock types are trivial to implement and we have done so as a proof-of-concept for a `pthread_mutex_t`. Their primary purpose is to adapt the different interfaces of the provided lock types to a common interface, which can be relied upon for the more advanced locks and abstractions provided by AthenaMP.

A nice side-effect of the lock adapters is that they turn the traditional C-style locks into more C++-style types. An example: it is a common mistake in OpenMP to forget to initialize a lock before using it (using `omp_init_lock`), or to forget to destroy it (using `omp_destroy_lock`) after it is needed. The *Resource Acquisition is Initialization*[1]

[a] http://www.thinkingparallel.com/2006/11/03/a-vision-for-an-openmp-pattern-library-in-c/

```
class omp_lock_ad {
public:
  omp_lock_ad();
  void set();
  void unset();
  int test();
  omp_lock_t& get_lock();
  ~omp_lock_ad();
};
```

Figure 1. Lock Adapter Interface

```
class account {
  omp_lock_ad lock;
  double balance;
};

void bank_transfer (account& send,
  account& recv, double amount)
{
  send.lock.set();
  recv.lock.set();

  send.balance -= amount;
  recv.balance += amount;

  recv.lock.unset();
  send.lock.unset();
}
```

Figure 2. A Bank Transfer with Lock Adapters

(*RAII*) idiom is employed to avoid the mistake: the locks are initialized in the constructor of the lock adapter and destroyed in the destructor. This ensures that locks are always properly initialized and destroyed without intervention from the programmer, even in the presence of exceptions. Should an exception be thrown and the thread leaves the area where the lock is defined, the lock adapter will go out of scope. As soon as this happens, its destructor is called automatically by the C++ runtime library, properly destroying the lock in the process. Thus, our lock adapters are exception-safe.

2.2 Scoped Locking with Guard Objects

The first generic lock type that builds on the lock adapters is called `guard`. It employs the *RAII*-idiom once again. This special case is so common that it has its own name: *scoped locking*[4]. A local, private guard object is passed a lock as a parameter in its constructor. The guard object sets the lock there and releases it when it goes out of scope (in its destructor). This way, it is impossible to forget to unset the lock (another common mistake when dealing with locks), even in the presence of multiple exit points or exceptions (as described in Sect. 2.1).

It is also possible to unset the lock directly (using `release`) and to acquire again later (using `acquire`), or to extract the lock out of the guard object (using `get_lock`). The interface of the guard objects is presented in Fig 3, along with a short example (a bank transfer again) in Fig. 4.

A guard object as implemented in AthenaMP can be instantiated with any locking class that provides the basic locking methods (i.e. `set`, `unset`, `test`). It does not depend on OpenMP in any way.

3 Deadlock Detection and Prevention

Deadlocks are an important and all-too common problem in multi-threaded code today. Consider the example code sketched in Fig. 2 again that shows how a bank transfer can

```
template <class LockType>
class guard {
public:
  guard (LockType& lock);
  void acquire();
  void release();
  LockType& get_lock();
  ~guard();
};
```

Figure 3. Guard Interface

```
void bank_transfer (account& send,
  account& recv, double amount)
{
  guard<omp_lock_ad> send_guard(send.lock);
  guard<omp_lock_ad> recv_guard(recv.lock);

  send.balance -= amount;
  recv.balance += amount;
}
```

Figure 4. A Bank Transfer with Guard Objects

be implemented. A deadlock might occur in this code, as soon as one thread performs a transfer from one account (let's call it account no. 1) to a different account (account no. 2), while another thread does a transfer the other way round (from account no. 2 to account no. 1) at the same time. Both threads will lock the sender's lock first and stall waiting for the receiver's lock, which will never become available because the other thread already owns it. More subtle deadlocks might occur, as soon as more accounts are involved, creating circular dependencies.

Sect. 3.1 describes a way to detect deadlocks semi-automatically, along with a corresponding implementation. Sect. 3.2 shows how to avoid deadlocks for an important subproblem.

3.1 Deadlock Detection using Levelled Locks

A common idiom to prevent deadlocks are lock hierarchies. If you always lock your resources in a predefined, absolute order, no deadlocks are possible. For our example, this means e.g. always locking account no. 2 before account no. 1. It can sometimes be hard to define an absolute order, though, a possible solution for part of this problem is described in Sect. 3.2.

Once a lock hierarchy for a project is defined, it may be documented in the project guidelines and developers are expected to obey it. Of course, there are no guarantees they will actually do so or even read the guidelines, therefore a more automated solution may be in order.

This solution is provided in the form of our second generic lock type: the `levelled_lock`. It encapsulates a lock adapter and adds a `_lock_level` to it, which is passed into the constructor of the class, associating a level with each lock. If a thread already holds a lock, it can only set locks with a lower (or the same - to allow for nested locks) level than the ones it already acquired. If it tries to set a lock with a higher level, a runtime exception of type `lock_level_error` is thrown, alerting the programmer (or quality assurance) that the lock hierarchy has been violated and deadlocks are possible.

The interface of the levelled locks and the bank transfer example are skipped here for brevity, as they are both very similar to the ones shown for the lock adapters, except of course for the additional `_lock_level`-parameter.

Like guard objects, a levelled lock as implemented in AthenaMP can be instantiated with any locking class that provides the basic locking methods (i.e. `set`, `unset`, `test`). Since the levelled locks provide these methods as well, guard objects can also be instantiated with them, thereby combining their advantages.

Our levelled locks have a configurable backend (via a template parameter) that actually stores the locks presently held for each thread. The first version we implemented depended on OpenMP, since their implementation uses threadprivate memory internally. This has also been a major implementation problem, because we found no OpenMP-compiler that could handle static, threadprivate containers — although judging from the OpenMP–specification this should be possible.

To fix this, a more generic backend was implemented. This version does not use threadprivate memory and does not depend on OpenMP. Instead it uses another generic component implemented in AthenaMP called `thread_storage`. The component stores all data in a vector that is indexed by a user-supplied thread-id. For OpenMP, this can be the number returned by `omp_get_thread_num`, for all other threading systems its the users responsibility to supply this id.

Both versions of locks with level checking induce a performance penalty. Furthermore, the use of throwing runtime exceptions in production code is limited. For this reason, a third backend for the levelled lock was implemented. It has the same interface as the other backends, but does no level checking. Of course, the basic locking functionality is provided. This class enables the programmer to switch between the expensive, checked version and the cheap, unchecked version with a simple `typedef`-command. This version of the levelled lock is called `dummy_levelled_lock` for the rest of this paper.

3.2 Deadlock Prevention using Dual-Guards / n-Guards

It has been shown in Sect. 3.1 that lock hierarchies are a very powerful measure against deadlocks. An argument that has been used against them in the past is that you may not be able to assign a unique number to all resources throughout the program. This is easy in our example of bank transfers, because every account most likely has a number and therefore this number can be used. It becomes more difficult when data from multiple sources (e.g. vectors, tables or even databases) need to share a single hierarchy.

While the general problem is difficult to solve, there is a solution for an important subclass: in our bank transfer example, two locks are needed at the same time for a short period. Even if there was no account number to order our locks into a hierarchy, there is another choice: although not every resource may have a unique number associated with it, every lock in our application does, since the lock's address remains constant during its lifetime.

One possible way to use this knowledge is to tell the user of our library to set their locks according to this implied hierarchy. But there is a better way: As has been described in Sect. 2.2, guard objects provide a convenient way to utilize exception-safe locking. Merging the idea presented above with guard objects results in a new generic locking type: the `dual_guard`. Its interface is sketched in Fig. 5 and our bank transfer example is adapted to it in Fig. 6.

No deadlocks are possible with this implementation, because the dual-guard will check the locks' addresses internally and make sure they are always locked in a predefined order (the lock with the higher address before the lock with the lower address to make them similar to the levelled locks as introduced in Sect. 3.1).

It is also obvious from this example, how the generic high-level lock types provided by AthenaMP raise the level of abstraction when compared to the more traditional locks

```
template <class LockType>
class dual_guard {
public:
  dual_guard(LockType&
    lock1, LockType& lock2);
  void acquire();
  void release();
  LockType& get_lock1();
  LockType& get_lock2();
  ~dual_guard();
};
```

Figure 5. Dual-Guard Interface

```
void bank_transfer (account& send,
  account& recv, double amount)
{
  dual_guard<omp_lock_ad>
    sr_guard(send.lock, recv.lock);

  send.balance -= amount;
  recv.balance += amount;
}
```

Figure 6. A Bank Transfer with Dual-Guards

offered by the common threading systems: to get the functionality that is provided with the one line declaration of the dual-guard, combined with an `omp_lock_ad`, two calls to initialize locks, two calls to destroy locks, two calls to set the locks and two calls to unset the locks are necessary, resulting in a total amount of eight lines of code. These eight lines of code are not exception-safe and possibly suffer from deadlocks, where the dual-guards are guaranteed to not have these problems.

As always, a dual-guard object as implemented in AthenaMP can be instantiated with any locking class that provides the basic locking methods (i.e. `set`, `unset`, `test`). It does not depend on OpenMP in any way. This also includes all variants of the levelled locks described in Sect. 3.1. When instantiated with a `levelled_lock`, using the lock's address to order them into a hierarchy is unnecessary, as there is a user-provided lock level inherent in these locks. In this case, the lock level is used for comparing two locks by the dual-guards automatically.

Generalizing the ideas presented in this section, we go one step further in AthenaMP and also provide an `n_guard` that takes an arbitrary number of locks. Since variadic functions are not well-supported in C++ and it is generally not recommended to use `varargs`, we have decided to let it take a `std::vector` of locks as argument. Apart from this, the functionality provided by these guard objects is equivalent to the dual-guards described above.

It should be noted again, that the dual-guards and n_guards presented in this section are only able to solve a subset of the general deadlock problem: only when two (or more) locks have to be set at the same point in the program, they can be employed. The solution is not applicable as soon as multiple locks have to be set at different times, possibly even in different methods, because the guards must be able to choose which lock to set first.

4 Performance

We carried out some really simple benchmarks to evaluate, how much the added features and safety impact the performance of locking. Table 1 shows how long it took to carry out a pair of set / unset operations for the respective locks. Table 2 shows the same data for the guard objects. All numbers are normalized over the course of 500.000 operations. To make the numbers for the dual-guards and n-guards comparable, we have also normalized those to one pair of operations by dividing by two or n respectively ($n = 20$ for the benchmark shown here).

Platform (Threads)	omp_lock_t	critical	omp_lock_ad	dummy_ll	ll
AMD (4)	0.40	0.38	0.56	0.67	1.48
SPARC (8)	1.02	1.47	1.09	1.56	4.96
IBM (8)	0.41	0.63	0.41	0.44	1.23

Table 1. Comparison of wall-clock times (in milliseconds) for one pair of lock/unlock operations.

Platform (Threads)	guard	dual_guard	n_guard (20 locks)
AMD (4)	0.65	0.43	0.26
SPARC (8)	1.51	0.87	0.37
IBM (8)	0.42	0.42	0.41

Table 2. Wall-clock times (in milliseconds) for one pair of lock/unlock operations for AthenaMP guards.

These tables show that there is a performance penalty associated with the added functionality. The levelled locks (shown as ll in the table) are the worst offenders and therefore the implementation of the dummy levelled lock (dummy_ll) does make a lot of sense.

5 Implementation Problems

The single most important problem when implementing the functionality described here is the present state of the compilers. Although we have tried various versions of different compilers, we have found not a single one that was able to compile all of our code and testcases, although we are quite sure they are valid according to both the C++ and OpenMP specifications. To name just a few of the problems we have encountered:

- Many compilers do not allow containers to be declared `threadprivate`.
- Local static variables cause problems with many compilers when declared `threadprivate`.
- Others have more or less subtle problems with exceptions when OpenMP is enabled (details can be found in a weblog entry of one of the authors[b]).

The functionality described here was implemented in merely about one thousand lines of code – yet, to convince ourselves that it is actually correct took a disproportionate amount of time. This convinced us that C++ and OpenMP is obviously not a combination that is wide-spread, at least judging from the state of these compilers.

6 Related Work and Contributions

An implementation of guard objects in C++ can be found in the `Boost.Threads` library[5], where they are called `boost::mutex::scoped_lock`. Their approach to the

[b] http://www.thinkingparallel.com/2007/03/02/exceptions-and-openmp-an-experiment-with-current-compiler s/

problem is different, though: Boost tries to be portable by providing different mutex implementations for different platforms. Our guard objects and high-level locks work on any platform, where a lock adapter can be implemented. Beyond that, this approach allows us to provide guard objects and advanced locking constructs on top of different lock variants, e.g. mutex variables, spinlocks, nested locks or others.

ZThreads[6] provides portable implementations for different lock types with C++, as well. The library includes guard objects that are instantiable with different lock types, but does not include our more advanced abstractions (e.g. levelled locks or dual-guards). It is a portable threading library, with focus on low level abstractions like condition variables and locks. AthenaMP, on the other hand, builds on OpenMP (which is already portable) and can therefore focus on higher-level components and patterns.

The idea of using lock hierarchies to prevent deadlocks is well-known[7]. The idea to automatically check the hierarchies has been described by Duffy for C#[8]. There is also a dynamic lock order checker called Witness available for the locks in the FreeBSD kernel[9].

As far as we know, none of the functionality described in this paper has been implemented with C++ and OpenMP. The idea of using memory addresses of locks to create a consistent lock hierarchy is also a novel contribution of this paper, as is the use of dual-guards (and n-guards) to hide the complexity of enforcing the lock hierarchy from the user.

7 Summary

Locks are still an important building block for concurrent programming, but the locks provided by most parallel programming systems are rather basic and error prone. In this paper, we have presented higher-level classes that encapsulate the locking functionality and provide additional value, in particular automatic lock hierarchy checking, automatic setting of multiple locks in a safe order, as well as exception-safety. We implemented our ideas in a generic way, decoupled from the parallel programming system used. In the future, we plan to implement other patterns with C++ and OpenMP, in the AthenaMP project.

References

1. B. Stroustrup, *The C++ Programming Language, Third Edition*, (Addison-Wesley 1997).
2. M. Suess, *AthenaMP*, `http://athenamp.sourceforge.net/`, (2006).
3. S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd edition, (Addison-Wesley, 2005).
4. D. C. Schmidt, M. Stal, H. Rohnert and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2 – Networked and Concurrent Objects*, (Wiley, 2000).
5. W. E. Kempf, *The boost.threads library*, `http://www.boost.org/doc/html/threads.html`, (2001).
6. E. Crahen, *Zthreads*, `http://zthread.sourceforge.net/`, (2000).
7. A. S. Tanenbaum, *Modern Operating Systems*, 2nd edition, (Prentice Hall, 2001).
8. J. Duffy, *No more hangs*, MSDN Magazine, **21**, (2006).
9. J. H. Baldwin, *Locking in the Multithreaded FreeBSD Kernel*, in: Proc. BSDCon, pp. 11–14, (2002).